1/73

Docket No. AT9-98-071

PATENT

#15 Appeal Brief 7.18.01 pc

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Re application of: **Blandy**	§	
	§	Group Art Unit: **2122**
Serial No.: **09/078,933**	§	
	§	Examiner: **Das, C.**
Filed: **May 14, 1998**	§	
	§	
For: **Method and Apparatus for Just in Time Compilation of Instructions**	§	

Assistant Commissioner for Patents
Washington, D.C. 20231

ATTENTION: Board of Patent Appeals and Interferences

Certificate of Mailing Under 37 C.F.R. § 1.8(a)
I hereby certify this correspondence is being deposited with the United States Postal Service as First Class mail in an envelope addressed to: Assistant Commissioner of Patents, Washington, D.C. 20231 on July 5, 2001.

By: Kate I. Campbell
Kate I. Campbell

APPELLANT'S BRIEF (37 C.F.R. 1.192)

This brief is in furtherance of the Notice of Appeal, filed in this case on February 5, 2001.

The fees required under § 1.17(c), and any required petition for extension of time for filing this brief and fees therefore, are dealt with in the accompanying TRANSMITTAL OF APPEAL BRIEF.

This brief is transmitted in triplicate. (37 C.F.R. 1.192(a))

REAL PARTIES IN INTEREST

The real party in interest in this appeal is the following party: International Business Machines of Armonk, New York.

RELATED APPEALS AND INTEREERENCES

With respect to other appeals or interference's that will directly affect, or be directly affected by, or have a bearing on the Board's decision in the pending appeal, there are no such appeals or interference's.

STATUS OF CLAIMS

A. TOTAL NUMBER OF CLAIMS IN APPLICATION

Claims in the application are: 1-32

B. STATUS OF ALL THE CLAIMS IN APPLICATION

1. Claims canceled: None
2. Claims withdrawn from consideration but not canceled: None
3. Claims pending: 1-32
4. Claims allowed: None
5. Claims rejected: 1-32

C. CLAIMS ON APPEAL

The claims on appeal are: 1-32

STATUS OF AMENDMENTS

A response to the Final Office Action was filed on December 15, 2000 in which Appellant proposed an Amendment to overcome the rejections of claims 1-32. The response included a proposed amendment to clarify some of the claim language. In response, an Advisory Action was mailed January 16, 2001, in which the Examiner states that the proposed amendment had been considered but was not deemed to place the application in condition for allowance.

SUMMARY OF INVENTION

The present invention provides a process in a data processing system for executing a method having a plurality of paths. Specification, page 6, lines 11-12. The data processing system executes native machine code. Specification, page 6, lines 12-13. A path is identified within the method that is being executed, wherein a plurality of bytecodes are associated with the path. Specification, page 6, lines 13-14. Bytecodes are compiled for the path being executed, wherein the bytecodes are compiled into native machine code for the data processing system, wherein bytecodes for unexecuted paths remain uncompiled. Specification, page 6, lines 14-17. Under the present invention, methods may be partially compiled, leaving infrequently executed paths within the methods in bytecode form, reducing storage use. Specification, page 6, lines 18-20. Compilation of bytecodes as a method is being interpreted resulting in just in time compilation of paths that are actually executed. Specification, page 6, lines 20-21. Bytecodes for a path that is compiled may be used during the same invocation of the method containing the path during which compilation of the bytecodes are being performed. Specification, page 6, lines 21-23.

ISSUES

The issue on appeal is whether:

claims 1-32 are anticipated under 35 U.S.C. 102(a) as being unpatentable over Kolawa (*Kolawa*), U.S. Patent No. 5,784,553 issued July 1998.

GROUPING OF CLAIMS

For purposes of this appeal, claims 1-32 do not stand or fall together as a single group. The claims are grouped in the following groups:

Group A contains claims 1, 4 and 17;

Group B contains claims 2, 5, 14, 18, 27 and 32;

Group C contains claims 3, 6, 13, 19, 26, and 31;

Group D contains claims 8, 15, 21 and 28;

Group E contains claims 9 and 22;

Group F contains claims 7, 10, 20 and 23;

Group G contains claims 11, 24 and 30;

Group H contains claims 12 and 25; and

Group I contains claims 16 and 29.

ARGUMENT

The Office Action rejects claims 1-32 under 35 U.S.C. § 102 as being anticipated by *Kolawa et al.* (U.S. Patent No. 5,784,553); hereinafter referred to as "*Kolawa*". This rejection is respectfully traversed.

Group A

The Office Action states in rejecting claims 1, 4 and 17:

> As per claim 1, 4 and 17, *Kolawa* et al teach the path is one of the path is shown in column 6 line 39-42 ("Each program is broken down into a series of code blocks comprising one or more program statements occurring along a single path of execution"), plurality of paths is shown in column 2 line 42-44 ("every branch at least once and optionally for generating as many paths as desired in the total path coverage set"), for the rest of the limitations see the rejection of the previous office action.

(Office Action dated October 3, 2000, pg. 2).

In rejecting claim 1, the Office Action further states:

> As per claim 1, Kolawa teaches **identifying a path within the method that is being executed** is shown in column 11 line 42-44 ("The dynamic symbolic execution is performing along the **path taken by an actual execution of the program"**), path taken by an actual execution of the program inherently including identifying a path within a method that is being executed, **plurality of instructions are associated with the paths** is shown in column 24 line 30-32 ("The TGS Driver Program executes the **program for the path** that corresponds to the input initially chosen and, **for all of the instructions found in the path**"), translating the first type instructions for the path being executed is shown in column 3 line 34-42 ("computer program written in the JAVA programming language, the **computer program being represented by JAVA bytecodes after being compiled by a JAVA compiler**. The method includes the steps of reading the JAVA bytecodes; obtaining an input value for the computer programs; **symbolically executing an instruction of the computer program represented**

in the JAVA bytecodes"), compiling JAVA program into **Java bytecode inherently including instructions are translated into second type instruction as claimed, unexecuted path remain untranslated** is shown in column 5 line 53-55 ("The original source code 11 comprises all types of files **used to express an uncompiled, that is, non-object code**, computer program, including definitional and declarative files"), **uncompiled that is non-object code** inherently including unexecuted path remain untranslated as claimed.

(Office Action dated March 2, 2000, pages 2-3).

With regard to independent claims 1, 4 and 17, these claims recite in their respective claim terminology, that instructions and bytecodes are processed only when a path containing these instructions or bytecodes is being executed while instructions and bytecodes associated with paths not being executed remain unprocessed. *Kolawa* does not teach this feature.

Claim 1 is reproduced below as being representative of the other rejected independent claims 4 and 17:

> 1. A process in a data processing system executing a routine having a plurality of paths, wherein the routine has includes a plurality of first type instructions and wherein the data processing system executes second type instructions, the process comprising:
> identifying a path being executed, wherein the path is one of the plurality of paths in the routine and wherein a plurality of first type instructions are associated with the path; and
> translating the first type instructions for the path being executed, wherein first type instructions are translated into second type instructions for execution by the data processing system, wherein first type instructions for unexecuted paths within the routine remain untranslated.

A prior art reference anticipates the claimed invention under 35 U.S.C. § 102 only if every element of a claimed invention is identically shown in that single reference, arranged as they are in the claims. *In re Bond*, 910 F.2d 831, 832, 15 U.S.P.Q.2d 1566, 1567 (Fed. Cir. 1990). All limitations of the claimed invention must be considered when determining patentability. *In re Lowry*, 32 F.3d 1579, 1582, 32 U.S.P.Q.2d 1031, 1034 (Fed. Cir. 1994). Anticipation focuses on whether a claim reads on the product or process a prior art reference discloses, not on what the reference broadly teaches. *Kalman v. Kimberly-Clark Corp.*, 713 F.2d 760, 218 U.S.P.Q. 781 (Fed. Cir. 1983). Appellant respectfully asserts that the Office Action has

not presented a case of anticipation because the reference, *Kolawa*, does not teach the claimed subject matter recited in independent claims 1, 4 and 17, as discussed hereafter.

Specifically, each and every feature of the presently claimed invention is not found arranged as they are in claim 1 of the present invention. The Office Action cites selected portions of *Kolawa* without regard to their actual arrangement in order to allege an anticipation rejection of claim 1 of the present invention. In alleging anticipation of the feature as recited in claim 1 "identifying a path being executed, wherein the path is one of the plurality of paths in the routine", the Office Action cites two unassociated sections of *Kolawa*. First, the Final Office Actions cites for the feature of "the path is one of the path":

Each program is broken down into a series of code blocks comprising one or more program statements occurring along a single path of execution.

(*Kolawa*, col. 6, lines 39-42).

Then the Office Action cites for the feature of "plurality of paths":

... every statement at least once and for taking substantially every branch at least once and optionally for generating as many paths as desired in the total path coverage set.

(*Kolawa*, col. 2, lines 42-44).

When taken together and analyzed, the two referenced sections of *Kolawa* actually do not teach identifying a path being executed, wherein the path is one of the plurality of paths in the routine as recited in claim 1 of the present invention. The first section of *Kolawa* referenced by the Office Action refers to a data structure for storing block and branch analysis information extracted from the original computer program (*Kolawa*, col. 6, lines 37-39). The second section of *Kolawa* referenced by the Office Action refers to finding a symbolic input for causing a program statement to execute (*Kolawa*, col. 2, lines 35-40). *Kolawa* teaches a system in which a computer program is run based on a symbolic execution. *Kolawa* does not teach identifying a path being executed. *Kolawa* symbolically executes each instruction in the program under test in a particular sequence (*Kolawa*, col. 13, lines 18-20). This sequence is not based on identifying a path but is based on execution of a set of inputs. *Kolawa* tests the program from the viewpoint of the most efficient use of the symbolic inputs and not identifying a path. *Kolawa* teaches that an input is found for causing a program element to execute in such a manner as to finding a minimal set of inputs for executing substantially every statement at least once (*Kolawa*, col. 2, lines 38-2). Only the inputs in *Kolawa* are identified and the paths are simply executed in a

sequential order without any regard to which path is executed. In fact, Kolawa does not always know which instructions is to be executed next. *Kolawa* states:

However, certain types of instructions cause a branch control flow and thereby make the next instruction unknown. For instance, conditional instructions, such as if or switch statements, or a loop condition test, such as in for or while statements, will cause a break in the control flow. Thus, if the next program instruction is unknown, (block 125), the branch condition and flow resolver 114 is called by the dynamic execution interpreter 110 to resolve the next instruction to be performed (block 126), as further described hereinbelow in FIG. 14.
(*Kolawa*, col. 13, lines 17-30).

Therefore, if the instruction to be symbolically executed is unknown, the path on which this instruction lies is also unknown.

Furthermore, when the program under test is substantially run through substantially all of the inputs is the desired coverage of the computer program considered adequate (*Kolawa*, col. 6, lines 1-3). The identity of the paths which are tested and even if all paths in the program are tested are no concern in *Kolawa* (only a substantial portion of the program need to be tested) (*Kolawa*, col. 9, lines 43-45). Therefore, *Kolawa* does not teach identifying the paths to be executed as recited in claim 1 of the present invention.

In addition, both of these referenced sections in *Kolawa* teach executing a program statement after identification of the program statement. First, a static symbolic execution of the program is performed for the complete program (*Kolawa*, col. 11, lines 26-27). This static symbolic execution consists of a single sweep over the full program and identifying branch conditions of program statements (*Kolawa*, col. 11, lines 27-32). In contrast, the present invention is directed to just-in-time compilation of instructions (Specification, page 6, lines 9-10). The Specification states:

> In particular, only paths that are actually being executed are being compiled or jitted. Additionally, the present invention allows for a path that has been compiled into native machine code to be used during the same invocation during which compilation is being performed.

(Specification, page 19, lines 15-19).

In the present invention methods are partially compiled, leaving infrequently executed

paths in bytecode form, thereby reducing storage use (Specification, page 20, lines 1-2). The present invention also provides cache efficiency by laying out the code in the order it is executed (Specification, page 20, lines 3-4). *Kolawa* does not teach leaving unexecuted paths in bytecode form or providing cache efficiency. *Kolawa* teaches providing for a symbolic input efficiency. *Kolawa* teaches that all instructions in a program are executed whether infrequently executed or not (*Kolawa*, col. 13, lines 18-20). All bytecodes in *Kolawa* are compiled when the bytecodes reach a virtual machine (*Kolawa*, col. 17, lines 60-63). All bytecodes in *Kolawa* are compiled because a test generation system driver program for symbolically executing instructions of the computer program represented by the bytecodes determines values of the program values at selected points of execution (*Kolawa*, col. 3, lines 24-28). The instructions are tested in such a way that all bytecodes have to be compiled at the start of the testing procedure because random values are provided to the program under test (*Kolawa*, col. 20, lines 19-21). Since random values are provided to the program under test, then all bytecodes are compiled at the start of the testing process.

Furthermore, for the translating step in claim 1, the Office Action references the following section of *Kolawa*:

> Yet another embodiment of the present invention is a computer-implemented method of finding inputs that will generate runtime errors in a computer program written in the JAVA programming language, the computer program being represented by JAVA bytecodes after being compiled by a JAVA compiler. The method includes the steps of reading the JAVA bytecodes; obtaining an input value for the computer program; symbolically executing an instruction of the computer program represented in bytecodes; ...

(*Kolawa*, col. 3, lines 34-42.)

The above reproduced section of *Kolawa* does not teach the translating step. This section, at most, teaches that a JAVA bytecode is read, obtains an input value and executes an instruction of the computer program represented in bytecodes by using the input value. This referenced section of *Kolawa* associates a bytecode instruction with an input value to simulate operation of the computer program instruction. No translating takes place. The bytecode is not converted from one form to another. The input value is not converted from one form to another. The input value is simply associated with the bytecode computer instruction to perform the program instruction. Therefore, *Kolawa* does not teach the step of translating the first type instructions for the path

being executed as recited in claim 1 of the present invention.

In addition, the Office Action references the following for the proposition that *Kolawa* teaches the feature of unexecuted paths remaining untranslated:

The original source code **11** comprises all types of files used to express an uncompiled, that is, non-object code, computer program, including definitional and declarative files. (*Kolawa*, col. 5, lines 53-55).

However, *Kolawa* further explains:

> For example, in the C programming language, header files are declarative files since they frequently declare but do not define program variable, structures and functions. Source code rules, however, are definitional files since they typically contain definitions of program variables, structures and functions.

(*Kolawa*, col. 5, lines 55-61).

Therefore, in context, what the referenced section of *Kolawa* states is that some files cannot be compiled because of their very nature, they contain only information about the program. This is not the same as the feature in the present invention as recited in claim 1 wherein first type instructions for unexecuted paths within the routine remain untranslated. The files referenced in *Kolawa* do not contain instructions and, hence, because of their very nature remain uncompiled whether they are being executed or not. In contrast, the present invention as recited in claim 1, instructions in unexecuted paths remain untranslated because the instructions remain unexecuted. Therefore, *Kolawa* does not teach the step of wherein first type instructions for unexecuted paths within the routine remain untranslated as recited in claim 1 of the present invention.

Further, *Kolawa* does not provide any teaching or suggestion of translating the instructions from one type of instruction to another type of instruction for a path being executed as recited in claim 1. The presently claimed invention translates instructions from one type into another as being executed. Based on the teaching of *Kolawa*, the translation of instructions occur through a compiling process, which does not involve execution of the program. Thus, when *Kolawa* is viewed as a whole for what it teaches, rather than in a piece meal fashion, each and every feature of the presently claimed invention is not shown in *Kolawa*, as arranged in claims 1, 4 and 17.

Furthermore, the section cited by the Office Action in regard to bytecodes are compiled into native machine code as recited in claim 4 states:

> The bytecodes are a relatively high-level representation of the source code so that some optimization and machine code generation (via a just-in-time compiler 214) can be performed at that level.

(*Kolawa*, col. 17, lines 51-54).

The bytecodes in *Kolawa* are not compiled into native machine code as recited in claim 4 of the present invention. The section cited by the Office Action merely states that bytecodes are optimized and machine code is generated at the level of the bytecodes. This section of *Kolawa* does not teach or even mention compiling bytecodes into native machine code. The just-in-time compiler does not perform any processes with the bytecode except for optimizing the bytecode. Simply because a just-in-time compiler does perform a separate and totally different process with the bytecode in *Kolawa*, does not mean that *Kolawa* teaches the bytecodes are compiled into native machine code as recited in claim 4.

In addition, the Office Action has not pointed out in *Kolawa* or any other prior art where identifying a path within the method that is being executed as recited in claims 4 and 17. However, for the same reasons as stated above in regard to claim 1, *Kolawa* does not teach identifying a path within the method being executed. *Kolawa* identifies an input in which to perform a symbolic execution of the program. *Kolawa* is concerned with the efficient use of these inputs and is not concerned with identifying a path being executed and does not teach identifying a path within the method being executed as recited in claims 4 and 17. *Kolawa* does not even concern itself with whether if every program statement is tested and considers an adequate coverage criteria if the program under test is run through substantially all of the inputs whether or not all program statements themselves are tested. The same goes for paths and paths within methods. *Kolawa* teaches that all inputs are input into the computer program and whatever program statements are affected by these inputs are of no concern.

Thus, *Kolawa* does not teach each and every feature recited in claims 1, 4 and 17 as is required under 35 U.S.C. § 102(a). Accordingly, Appellant respectfully asserts that the rejection of claims 1, 4 and 17 under 35 U.S.C. §102(a) has been overcome.

Furthermore, there is no teaching or suggestion in *Kolawa* to modify its teachings to arrive at Appellant's claimed invention. There is no teaching or suggestion in *Kolawa* for

translating the first type instructions for the path being executed or translating the first type instructions for the path being executed, wherein first type instructions are translated into second type instructions for execution by the data processing system, wherein first type instructions for unexecuted paths within the routine remain untranslated because *Kolawa* requires the translation of instructions occur through a compiling process, which does not involve execution of the program. In addition there is no teaching or suggest in Kolawa for identifying a path or identifying a path within the method as recited in claims 1, 4 and 17. *Kolawa* is concerned solely with the inputs of the program in which the symbolic execution is being performed and does not teach, suggest or even mention identifying a path or a path within the method.

Furthermore, absent some teaching or incentive to implement *Kolawa* in which to identify the paths or paths within the methods being executed, one of ordinary skill in the art would not be led to modify *Kolawa* to reach the present invention when the reference is examined as a whole. Absent some teaching, suggestion, or incentive to modify *Kolawa* in this manner, the presently claimed invention can be reached only through an improper use of hindsight using the Appellant's disclosure as a template to make the necessary changes to reach the claimed invention.

Group B

With regard to claim 2, being representative of claims 5, 14, 18, 27 and 32, *Kolawa* does not teach executing second type instructions for a path in response to a loop back through the path during execution of the routine. In rejecting claim 2, the Office Action states:

> Kolawa et al teach executing second type instructions for a path in response to a loop back through the path is shown in column 22 line 56-61 ("The SVM then gets the next JAVA bytecode (second type instruction) from the .class files (block 472). If all JAVA program instructions have not been symbolically executed (block 428), control returns to the top of the control loop (block 421) for processing of the next program instruction"), during execution of the routine in column 26 line 41-43 ("A normal JAVA virtual machine can be used to execute the instrumented bytecodes").

(Office Action dated March 2, 2000, page 3).

Respectfully, the Office Action is taking a mere statement "returns to the top of the control loop" in *Kolawa* and equating this statement to a specific feature in claims 2, 5, 14, 18,

27 and 32 of the present invention. The control loop referred to in *Kolawa* is in reference to a flow diagram of a method for performing symbolic execution using the Symbolic Virtual Machine of FIG. 19 of the present invention (*Kolawa*, col. 22, lines 32-35) and does not teach a path in response to a loop back through the path during execution of the routine as recited in claim 2. The control loop taught in *Kolawa* is in response to not symbolically executing a program instruction and has nothing to do with executing instructions for a path as recited in claim 2. In fact, once the control loop referenced in *Kolawa* is reached and all instructions have been processed the operation ends and does not return to process another instruction. *Kolawa's* teaching is directly opposed to the present invention as recited in claim 2. Similarly, *Kolawa* does not teach executing a executing native machine code for a path in response to a loop back through the path during execution of the method as recited in claim 5 or executing the instructions for a path within the plurality of paths in response to a loop back through the path during compilation of the method as recited in claim 14. As stated above, *Kolawa* teaches the execution of inputs in order to symbolically execute a program. *Kolawa* does not teach executing instructions, native machine code or instructions for a path within a plurality of paths. *Kolawa* teaches that program instructions are merely symbolically interpreted and does not teach executing native machine code but only teaches that native machine code is generated. Furthermore, Kolawa does not teach executing instructions for a path within the plurality of paths.

Therefore, *Kolawa* does not teach each and every feature as recited in claims 2, 5, 14, 18, 27 and 32, as is required under 35 U.S.C. § 102. Accordingly, Appellant respectfully requests withdrawal of the rejection of claims 2, 5, 14, 18, 27 and 32 under 35 U.S.C. § 102.

Group C

With regard to claim 3, being representative of claims 6, 13, 19, 26 and 31, *Kolawa* does not teach translated instructions for the path are executed in an order and wherein the translated instructions are stored in execution order. In rejecting claim 3, the Office Action states:

> Kolawa et al teach translated instruction for the path are executed in an order is shown in column 25 line 60-64 (" the TGS Driver Program read the JAVA Bytecodes (translated instruction) in the .class files for the module under test at step 480. The Module Testing Module 238 at step 482 decides on a first

sequence of methods to be executed"), sequence of methods to be executed inherently including translated instructions are stored in an execution order.

(Office Action dated March 2, 2000, page 3).

The Office Action's cited section of *Kolawa* allegedly teaching the present invention in claim 3 states:

> First, the TGS Driver Program reads the JAVA bytecodes in the .class files for the module under test at step 480. The Module Testing Module 238 at step 482 decides on a first sequence of method to be executed (for example, a random sequence or calling each method mode) and the arguments to be passed to them.

(*Kolawa*, col. 25, lines 60-66).

This section of *Kolawa* does not teach the translated instructions are stored in execution order. In fact, as stated above in regard to claim 1, *Kolawa* does not provide any teaching or suggestion of translating instructions from one type of instruction to another type of instruction for a path being executed. Furthermore, *Kolawa* does not teach the translated instructions are stored in execution order. While *Kolawa* does store symbolic memory values (*Kolawa*, col. 14, lines 44-45), this does not equate to storing translated instructions in execution order. Kolawa does not store instructions in execution order but relies on execution of the inputs to determine if substantially all of the program has been tested. The system in Kolawa does not know in which order the instructions were symbolically executed. Hence, Kolawa does not teach storing the instructions in execution order.

Therefore, *Kolawa* does not teach each and every feature as recited in claims 3, 6, 13, 19, 26, and 31, as is required under 35 U.S.C. § 102. Accordingly, Appellant respectfully requests withdrawal of the rejection of claims 3, 6, 13, 19, 26, and 31 under 35 U.S.C. § 102.

Group D

With regard to claim 8, being representative of claims 15, 21 and 28, *Kolawa* does not teach a data structure is used during compiling of the method to store information about a path as the path is compiled. In rejecting claim 8, the Office Action states:

> Kolawa et al teach data structure is used during compiling of the method store information about a path as claimed is shown in column 6 line 37-42 ("FIG. 2D is a data structure for storing block and branch analysis information extracted from the original computer program. Each program is broken down into a series

of code blocks comprising one or more program statements occurring along a single path of execution").

(Office Action dated March 2, 2000, page 5).

The Office Action's cited section of *Kolawa* allegedly teaching the present invention in claim 8 states:

> FIG 2D is a data structure for storing block and branch analysis information extracted from the original computer program. Each program is broken down into a series of code blocks comprising one or more program statements occurring along a single path of execution. A branch condition causes a break in flow and results in two or more code blocks being formed.

(*Kolawa*, col. 6, lines 37-42).

While this section of *Kolawa* does mention data structures, this alone does not teach the present invention as recited in claim 8. The mere mention of a data structure does not teach using the data structure during compiling of the method to store information about a path as the path is compiled. First, the data structure in Kolawa stores block and branch analysis information extracted from the computer program and not information about a path as the path is compiled. Second the data structure in Kolawa stores block and branch information only after analyzing the computer program and not during a compiling process as recited in claim 8 of the present invention. The features of claim 8 exemplify the differences between the present invention and *Kolawa*. As stated above, the present invention concerns executing instructions and performing actions regarding these executed instructions on a just in time basis. In contrast, the system taught in *Kolawa*, first sweeps the entire program under test and only then performs the symbolic execution of the program. The system in *Kolawa* includes a multitude of steps for each instruction within the program while the present invention translates an instruction and then moves on to the next instruction without having to return to the previous instruction. The differences in the present invention and *Kolawa* is clear as indicated by dependent claim 8. *Kolawa* performs actions on program instructions after first analyzing the entire program and the present invention performs these actions on a just in time basis without having to first analyze the instructions.

Therefore, *Kolawa* does not teach each and every feature as recited in claims 8, 15, 21, and 28, as is required under 35 U.S.C. § 102. Accordingly, Appellant respectfully requests withdrawal of the rejection of claims 8, 15, 21, and 28 under 35 U.S.C. § 102.

Group E

With regard to claim 9, being representative of claim 22, *Kolawa* does not teach the data structure stores the native machine code. In rejecting claim 9, the Office Action states:

> Kolawa et al teach data structure stores the native machine codes is shown in column 6 lines 37-42 and column 51-54 ("The bytecodes are a relatively high-level representation of the source code so that some optimization and machine code generation (via a just-in-time compiler 214) can be performed at that level") and ("FIG. 2d is a data structure for storing block and branch analysis information extracted from the original computer program").

(Office Action dated March 2, 2000, page 5).

These two sections of *Kolawa* have been used by the Office Action to allege a rejection of other distinctly different features of the present. In particular, the section of *Kolawa* allegedly teaching storing native machine codes (*Kolawa*, col. 6, lines 37-42) is also used by the Office Action to allege the teaching of the features of claim 8. However, as explained above in regard to claim 8, this section of *Kolawa*, furthermore, does not teach the present invention as recited in claim 9. In particular, this cited section does not teach a data structure storing machine code. As stated above, *Kolawa* merely generates machine codes, but there is no teaching, suggestion or even mention in *Kolawa* of storing this machine code.

Furthermore, for the feature of storing native machine code in the data structure, the Office Action cites the section of *Kolawa* which states:

> The bytecodes are a high-level representation of the source code so that some optimization and machine code generation (via a just-in-time compiler 214 can be performed at that level.

(*Kolawa*, col. 17, lines 51-54).

This section does not teach storing and, furthermore, does not teach storing native machine code. Respectfully, the Office Action is taking unrelated parts of *Kolawa* in order to allege a rejection of the present invention when, if fact, these sections do not teach the present invention.

Therefore, *Kolawa* does not teach each and every feature as recited in claims 9 and 22, as is required under 35 U.S.C. § 102. Accordingly, Appellant respectfully requests withdrawal of the rejection of claims 9 and 22 under 35 U.S.C. § 102.

Group F

With regard to claims7, 10, 20 and 23, being representative of claims 10, 20 and 23, *Kolawa* does not teach a JIT station is used in compiling the method. In rejecting claim 7, the Office Action states:

> Kolawa et al teach JIT in compiling method is shown in column 17 line 62-65 ("optionally turned into machine code by the Just-In-Time Compiler 214. The JAVA Interpreter and Just-In-Time Compiler operate in the context of a Runtime System 222).

(Office Action dated March 2, 2000, page 5).

As with the alleged rejection of claims 9 and 22 above, the Office Action is citing sections of *Kolawa* and alleging that these section teach multiple features of the present invention when in fact, these sections do not teach, suggest, or even mention the features of the present invention. In particular, the cited section of *Kolawa*, col. 17, lines 62-65, which has been used to allege a teaching of a data structure storing the native machine code, is also being used to allege that not only is the machine code stored but a JIT station is used in compiling the method. However, this section of Kolawa, and Kolawa as a whole does not teach a JIT station or using a JIT station in compiling the method. As stated above, Kolawa is not concerned with just-in-time execution of a program. Kolawa teaches a system in which en entire program is analyzed and then symbolically executed. When the entire program is analyzed in Kolawa, there is no just-in-time analyzing but instead there is a single sweep over the entire program to generate expressions for path conditions (Kolawa, col. 11, lines 26-30). The section cited by the Office Action does not teach compiling the method and does not teach using a JIT station to compile the method as recited in claim 7 of the present invention.

Therefore, *Kolawa* does not teach each and every feature as recited in claims 7, 10, 20 and 23, as is required under 35 U.S.C. § 102. Accordingly, Appellant respectfully requests withdrawal of the rejection of claims 7, 10, 20 and 23 under 35 U.S.C. § 102.

Group G

With regard to claim 11, being representative of claims 24 and 30, *Kolawa* does not teach identifying the method that is to be executed and compiling the bytecodes into instructions for execution by the data processing system for each path within the plurality of paths as each path is executed. In rejecting claim 11, the Office Action states:

> Kolawa et al teach identifying the method that is to be executed is shown in column 20 line 36-38 ("Module 41 identifies the branch condition controlling the execution of that code block"), compiling the bytecodes is shown in column 26 line 44-47 ("FIG. 23 is a flow diagram of the steps for handling JAVA bytecode instrumentation. At step 492, the JAVA parser 230 reads the .class files 210 generated by the JAVA Compiler 208 and the .Java files 200 of the original source code").

(Office Action dated March 2, 2000, page 6).

The Office Action's cited section of *Kolawa* allegedly teaching the present invention in claim 11 states:

> Otherwise, if the selected code block is not covered (block 255), the Decision Module 41 identifies the branch condition controlling the execution of that code block (block 256).

(*Kolawa*, col. 20, lines 36-38).

In addition, the section cited by the Office Action in regard to compiling the bytecodes as recited in claim 11:

> FIG. 23 is a flow diagram of the steps for handling JAVA bytecode instrumentation. At step 492, the JAVA parser 230 reads the .class files 210 generated by the JAVA Compiler 208 and the .java files 200 for the original source code.

(*Kolawa*, col. 26, lines 44-47).

JAVA bytecode instrumentation in *Kolawa* does not compile the bytecode as recited in claim 11 of the present invention. JAVA bytecode instrumentation is used for profiling code by adding bytecodes that keep track of what portions of the code get executed (*Kolawa*, col. 26, lines 22-24). A JAVA virtual machine can be used to execute the instrumented bytecodes in order to gather extra information about the JAVA program under test (*Kolawa*, col. 26, lines 41-43). In

contrast, JAVA bytecode instructions are generated that are non-specific to a particular computer architecture (Specification, pg. 4, lines 8-9). The section referenced by the Office Action allegedly teaching compiling the bytecodes merely reads a file. There is no teaching of compiling of bytecodes in this referenced section of *Kolawa*. Furthermore, the Office Action has not pointed out in *Kolawa* or any other prior art where compiling the bytecodes into instructions is taught as recited in claim 11. In addition, the Office Action has not pointed out in *Kolawa* or any other prior art where compiling the bytecodes into instructions for each path as the path is executed as recited in claim 11.

Thus, *Kolawa* does not teach each and every feature recited in claims 11, 24 and 30, as is required under 35 U.S.C. § 102. Accordingly, Appellant respectfully requests withdrawal of the rejection of claims 11, 24 and 30 under 35 U.S.C. § 102.

Group H

With regard to claim 12, being representative of claim 25, *Kolawa* does not teach unexecuted paths within the plurality of paths remain in a bytecode form. In rejecting claim 12, the Office Action states:

> Kolawa et al teach unexecuted paths remain in bytecode form is shown in column 5 line 53-55 ("The original code 11 comprises all types of files used to express an uncompiled, that is, non-object code, computer program, including definitional and declarative files"), uncompiled that is non-object code inherently including unexecuted paths as claimed.

(Office Action dated March 2, 2000, page 6).

The Office Action's cited section of *Kolawa* allegedly teaching the present invention in claim 12 states:

> The original source code 11 comprises all types of files used to express an uncompiled, that is, non-object code, computer program, including definitional and declarative files.

(*Kolawa*, col. 5, lines 53-55).

While the Office Action alleges that uncompiled non-object code inherently includes unexecuted paths, the Office Action fails to address the additional features of claim 12 in which

the unexecuted paths within the plurality of paths remain in bytecode form. This section of *Kolawa* does not teach that the unexecuted paths within the plurality of paths remain in bytecode form. This section of *Kolawa* does not teach or even mention that the paths are in bytecode form. Again, as stated above in regard to the rest of the claims in the present invention, the Office Action is, respectfully, making general rejections for specific features of the claims in the present invention. In particular, in claim 12, the Office Action alleges that unexecuted paths within the plurality of paths remain in bytecode form without any indication where in *Kolawa* or any other prior art where this feature is taught. Anticipation focuses on whether a claim reads on the product or process a prior art reference discloses, not on what the reference broadly teaches. *Kalman v. Kimberly-Clark Corp.*, 713 F.2d 760, 218 U.S.P.Q. 781 (Fed. Cir. 1983). While *Kolawa* may broadly teach uncompiled source code, *Kolawa* does not teach that unexecuted paths remain in bytecode form as recited in claim 12 of the present invention.

Thus, *Kolawa* does not teach each and every feature recited in claims 12 and 25, as is required under 35 U.S.C. § 102. Accordingly, Appellant respectfully requests withdrawal of the rejection of claims 12 and 25 under 35 U.S.C. § 102.

Group I

With regard to claim 16, being representative of claim 29, *Kolawa* does not teach the data structure stores the instructions. In rejecting claim 16, the Office Action states:

> Kolawa teach data structure stores the instructions is shown in column 4 lines 36-38 ("FIG. 2D is a data structure for block and branch analysis information stored in the program database of FIG. 2A").

(*Kolawa*, col. 4, lines 36-38).

As with claim 12 above, and with the alleged rejections of claims 1-32 of the present invention overall, the Office Action is again using a general teaching of *Kolawa*, to allege a rejection of claim 16. The section of *Kolawa* cited by the Office Action allegedly rejecting claim 16 describes a Figure 2D in *Kolawa*. This description of Figure 2D does not teach the features of claim 16. This description of Figure 2D does not teach that the data structures stores the instructions but merely teaches that Figure 2D is a data structure which stores block and branch information. However, the description of Figure 2D does not teach the data structure storing instructions. The block and branch information referenced in *Kolawa* are not instructions

but are instructions extracted from the original computer program. Each program is broken down into blocks and a branch condition causes a break in execution flow of the program (*Kolawa*, col. 6, lines 37-42). These blocks and branch conditions in *Kolawa* are not the same as the instructions recited in claim 16. The Specification states:

> PSI buffer 410 contains the output, in the form of machine code instructions, from jitting. The instructions may be stored within PSI buffer 410 temporarily until they are stored elsewhere within the data processing system, such as another buffer.

(Specification, page 12, lines 23-25).

The machine code instructions referenced in claim 16 and the block and branch information taught by *Kolawa* are in opposite. The block and branch condition information is extracted from the computer program under test in *Kolawa* while the machine code instructions in the present invention are output from a buffer in response to jitting. As explained above, *Kolawa* does not teach a jitting method. *Kolawa* merely teaches that after a single sweep of the entire program, the program is symbolically executed. No jitting is involved in Kolawa.

Thus, *Kolawa* does not teach each and every feature recited in claims 16 and 29, as is required under 35 U.S.C. § 102. Accordingly, Appellant respectfully requests withdrawal of the rejection of claims 16 and 29 under 35 U.S.C. § 102.

Conclusion

Based on the above, Appellant respectfully submits that all of the claims stand in condition for allowance and that the rejections set forth in the Office Action have been overcome. Accordingly, Appellant respectfully requests that the Board of Patent Appeals and Interferences reverse the decision of the Examiner and allow all of the pending claims.

Charles D. Stepps, Jr.
Reg. No. 45,880
Carstens, Yee & Cahoon, LLP
PO Box 802334
Dallas, TX 75380
(972) 367-2001

APPENDIX OF CLAIMS

The text of the claims involved in the appeal are:

1. (Amended) A process in a data processing system executing a routine having a plurality of paths, wherein the routine has includes a plurality of first type instructions and wherein the data processing system executes second type instructions, the process comprising:

identifying a path being executed, wherein the path is one of the plurality of paths in the routine and wherein a plurality of first type instructions are associated with the path; and

translating the first type instructions for the path being executed, wherein first type instructions are translated into second type instructions for execution by the data processing system, wherein first type instructions for unexecuted paths within the routine remain untranslated.

2. The process of claim 1 further comprising:

executing second type instructions for a path in response to a loop back through the path during execution of the routine.

3. The process of claim 1, wherein translated instructions for the path are executed in an order and wherein the translated instructions are stored in execution order.

4. (Amended) A process in a data processing system for executing a method having a plurality of paths, wherein the data processing system executes native machine code, the process comprising:

identifying a path being executed, wherein the path is one of the plurality of paths in the within the method and wherein a plurality of bytecodes are associated with the path; and

compiling bytecodes for the path being executed, wherein the bytecodes are compiled into native machine code executed by the data processing system, wherein bytecodes for unexecuted paths within the method remain uncompiled.

5. The process of claim 4 further comprising:

executing native machine code for a path in response to a loop back through the path during execution of the method.

6. The process of claim 4, wherein compiled instructions for the path are executed in an order and wherein the compiled instructions are stored in execution order.

7. The process claim 4, wherein a JIT station used is in compiling the method.

8. The process of claim 4, wherein a data structure is used during compiling of the method to store information about a path as the path is compiled.

9. The process of claim 8, wherein the data structure stores the native machine code.

10. The process of claim 9, wherein the data structure is a JIT station.

11. A process in a data processing system for executing a method having a plurality of paths in which each path with in the plurality of paths contains a number of bytecodes, the method comprising:

identifying the method that is to be executed; and

compiling the bytecodes into instructions for execution by the data processing system for each path within the plurality of paths as each path is executed.

12. The process of claim 11, wherein unexecuted paths within the plurality of paths remain in a bytecode form.

13. The process of claim 11, wherein the instructions have an execution order and further comprising:

storing the instructions in the execution order.

14. The process of claim 11 further comprising:

executing the instructions for a path within the plurality of paths in response to a loop back through the path during compilation of the method.

15. The process of claim 11, wherein a data structure is used during compiling of the method to store information about a path as the path is compiled.

16. The process of claim 15, wherein the data structure stores the instructions.

17. A data processing system for executing a method having a plurality of paths, wherein the data processing system executes native machine code, the data processing system comprising:

identification means for identifying a path being executed, wherein the path is one of the plurality of paths in the method and wherein a plurality of bytecodes are associated with the path; and

compilation means for compiling bytecodes for the path being executed, wherein the bytecodes are compiled into native machine code, wherein bytecodes for unexecuted paths remain uncompiled. with the method

18. The data processing system of claim 17 further comprising:

execution means for executing native machine code for a path in response to a loop back through the path during interpreting of the method.

19. The data processing system of claim 17, wherein compiled instructions for the path are executed in an order and wherein the compiled instructions are stored in the execution order.

20. The data processing system of claim 17, wherein a JIT station used is in compiling the method.

21. The data processing system of claim 17, wherein a data structure is used during compiling of the method to store information about a path as the path is compiled.

22. The data processing system of claim 21, wherein the data structure stores the native machine code.

23. The data processing system of claim 22, wherein the data structure is a JIT station.

24. A data processing system comprising:

a method having a plurality of paths in which each path within the plurality of paths contains a number of bytecodes;

identification means for identifying that the method is to be executed; and

compilation means for compiling the bytecodes into instructions for execution by the data processing system for each path within the plurality of paths as each path is executed.

25. The data processing system of claim 24, wherein unexecuted paths within the plurality of paths remain in a bytecode form.

26. The data processing system of claim 24, wherein the instructions have an execution order and further comprising:

storing means for storing the instructions in the execution order.

27. The data processing system of claim 24 further comprising:

execution means for executing the instructions for a path within the plurality of paths in response to a loop back through the path during compilation of the method.

28. The data processing system of claim 24, wherein a data structure is used during compiling of the method to store information about a path as the path is compiled.

29. The data processing system of claim 28, wherein the data structure stores the instructions.

30. A computer program product for executing a method in a data processing system, wherein the method has a plurality of paths in which each path with in the plurality of paths contains a number of bytecodes, the computer program product comprising:

first instructions for identifying that the method is to be executed; and

second instructions for compiling the bytecodes into compiled instructions for execution by the data processing system for each path within the plurality of paths as each path is executed.

31. The computer program product of claim 30, wherein the compiled instructions have an execution order and further comprising:

third instructions for storing the compiled instructions in the execution order.

32. The method of claim 30 further comprising:

third instructions for executing the compiled instructions for the path within the plurality of paths in response to a loop back through the path during compilation of the method.